Reference Manual

For Audio Processing

API Version 1.15

LEGAL DISCLAIMER

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site.

MPEG is an international standard for video compression/decompression promoted by ISO. Implementations of MPEG CODECs, or MPEG enabled platforms may require licenses from various entities, including Intel Corporation.

Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel.

Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Table of Contents

Overview

This document describes API for audio processing. The API is similar to video processing API implemented in video library from **Intel® Media SDK for Windows*** or **Intel® Media Server Studio * – SDK** (hereinafter **SDK**)**.**

The API is implemented in **Audio for Windows*** and **Intel® Media Server Studio * - Audio Encoder & Decoder (**hereinafter **Audio**)**.**

Document Conventions

The API uses the Verdana typeface for normal prose. With the exception of section headings and the table of contents, all code-related items appear in the `Courier New` typeface (`mxfStatus` and `MFXInit)`. All class-related items appear in all cap boldface, such as **DECODE** and **ENCODE**. Member functions appear in initial cap boldface, such as **Init** and **Reset**, and these refer to members of all three classes, **DECODE**, **ENCODE** and **VPP**. Hyperlinks appear in underlined boldface, such as **`mfxStatus`**.

Acronyms and Abbreviations

MP3	MPEG-1 Audio Layer 3
AAC	Advanced Audio Coding

Related Documents

SDK Reference manual	
SDK API Reference manual (Extensions for User-Defined Functions)	

Architecture

Audio library supports next functionality:

DECODE	Decode compressed audio streams into raw samples
ENCODE	Encode raw audio samples into compressed bitstreams
CORE	Auxiliary functions for synchronization
USER	User-defined functions for plugins loading (**New!**)
Misc	Global auxiliary functions

With the exception of the global auxiliary functions, **SDK** and **Audio** functions are named after their functioning domain and category, as illustrated in Figure 1.

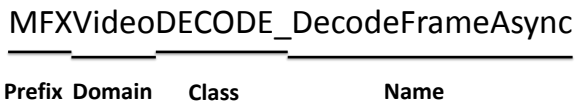

Figure 1: SDK Function Naming Convention

Applications use **Audio** functions by linking with the **SDK** dispatcher library, as illustrated in Figure 2. The dispatcher library identifies the most suitable library, and then redirects function calls.

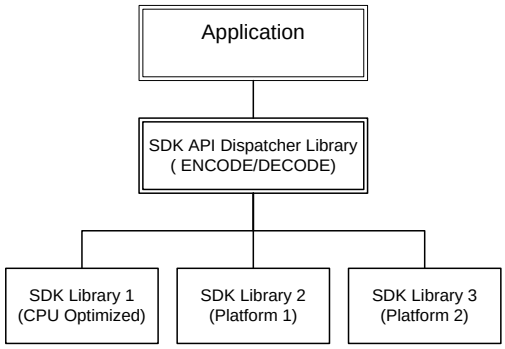

Figure 2: SDK Dispatching Mechanism

Audio Data

Audio processes audio data by small chunks of samples also known as audio frames. One frame of audio data consists of predefined by standard number of audio samples. If audio

stream consists of more than one channel, all channels are interleaved in the same audio frame. Number and order of channels in the data buffers are defined by the **Audio** components configuration.

Audio uses two different data structures to hold audio frames, mfxBitstream, that is used for compressed audio data and mfxAudioFrame that holds raw audio samples. Both structures may hold partial audio frame, complete frame or several audio frames.

Term frame is used for both compressed and uncompressed audio data.

Audio Decoding

The **DECODE** class of functions takes a compressed bitstream as input and converts it to audio samples as output.

DECODE processes only pure or elementary audio streams. The library cannot process bitstreams that reside in a container format, such as MP4 or MPEG. The application must first de-multiplex the bitstreams. De-multiplexing extracts pure audio streams out of the container format.

For MP3 standard the application can provide the input bitstream as one complete frame of data, less than one frame (a partial frame), or multiple frames. If only a partial frame is provided, **DECODE** internally constructs one frame of data before decoding it. For AAC standard, **DECODE** accepts only complete audio frame.

The time stamp of a compressed buffer must be accurate to the first byte of the frame data. This time stamp will be assigned to uncompressed audio frame at decoder output and later may be used for audio video synchronization.

DECODE supports repositioning of the bitstream at any time during decoding. The application should use Reset function before starting decoding from new position to clear internal decoder history.

Audio Encoding

The **ENCODE** class of functions takes audio samples as input and compresses them into a bitstream.

An encoder may receive partial frame as input, complete frame or several frames in the same input data buffer.

An **ENCODE** output consists of compressed audio frame with correspondent time stamp. Encoder uses timestamp provided by the application together with input audio samples. The time stamp is used for multiplexing audio and video. **Audio** library provides only pure audio stream encoding. The application must provide its own multiplexing.

Audio Processing

Audio does not support audio processing like sampling rate conversion, denoising and so on.

Custom plugins support (**New!**)

New API for custom codec plugins support was added to **Audio** library starting from API 1.13. A set of new API functions allows a user to add custom codec support to audio transcoding pipeline. Plugins architecture, API and a guide for audio plugins creation is described in *SDK API Reference Manual (Extensions for User-Defined Functions)*.

Programming Guide

This chapter describes the concepts used in programming the API.

The application must use the include file, **mfxaudio.h** (for C programming), or **mfxaudio++.h** (for C++ programming), and link the **SDK** static dispatcher library, libmfx.lib.

```
Include these files:
        #include "mfxaudio.h"        /* The SDK include file */
        #include "mfxaudio++.h"      /* Optional for C++ development */
Link this library:
        libmfx.lib                   /* The SDK static dispatcher library */
```

Status Codes

Audio functions organize into classes for easy reference. The classes include **ENCODE** (encoding functions) and **DECODE** (decoding functions).

Init, **Reset** and **Close** are member functions within the **ENCODE** and **DECODE** classes that initialize, restart and de-initialize specific operations defined for the class. Call all other member functions (except **Query** and **QueryIOSurf**) within the **Init** … **Reset** (optional) … **Close** sequence.

The **Init** and **Reset** member functions both set up necessary internal structures for media processing. The difference between the two is that the **Init** functions allocate memory while the **Reset** functions only reuse allocated internal memory. Therefore, **Reset** can fail if **Audio** needs to allocate additional memory. **Reset** functions can also fine-tune **ENCODE** parameters during processing or reposition a bitstream during **DECODE**.

All **Audio** functions return status codes to indicate whether an operation succeeded or failed. See the **mfxStatus** enumerator description for all defined status codes. The status code **MFX_ERR_NONE** indicates that the function successfully completed its operation. Status codes are less than **MFX_ERR_NONE** for all errors and greater than **MFX_ERR_NONE** for all warnings.

If an **Audio** function returns a warning, it has sufficiently completed its operation, although the output of the function might not be strictly reliable. The application must check the validity of the output generated by the function.

If an **Audio** function returns an error (except **MFX_ERR_MORE_DATA** or **MFX_ERR_MORE_BITSTREAM**), the function aborts the operation. The application must call either the **Reset** function to put the class back to a clean state, or the **Close** function to terminate the operation. The behavior is undefined if the application continues to call any class member functions without a **Reset** or **Close**. To avoid memory leaks, always call the **Close** function after **Init**.

Audio Session

Before calling any **Audio** functions, the application must initialize **Audio** library and create an **Audio** session. An **Audio** session maintains context for the use of any of **DECODE** and **ENCODE** functions.

The function **MFXInit** starts (initializes) a session. **MFXClose** closes (de-initializes) the session. To avoid memory leaks, always call **MFXClose** after **MFXInit**.

Multiple Sessions

Each **Audio** session can run exactly one instance of **DECODE** or **ENCODE** functions. This is good for a simple transcoding operation. If the application needs more than one instance of **DECODE** or **ENCODE** in a complex transcoding setting, or needs more simultaneous transcoding operations, the application can initialize multiple sessions.

The application can use multiple sessions independently or run a "joined" session. Independently operated sessions cannot share data unless the application explicitly synchronizes session operations (to ensure that data is valid and complete before passing from the source to the destination session.)

To join two sessions together, the application can use the function **MFXJoinSession**. Alternatively, the application can use the function **MFXCloneSession** to duplicate an existing session. Joined sessions work together as a single session, sharing all session resources, threading control and prioritization operations. When joined, one of the sessions (the first join) serves as a parent session, scheduling execution resources, with all others child sessions relying on the parent session for resource management.

With joined sessions, the application can set the priority of session operations through the **MFXSetPriority** function. A lower priority session receives less CPU cycles.

After the completion of all session operations, the application can use the function **MFXDisjoinSession** to remove the joined state of a session. Do not close the parent session until all child sessions are disjoined or closed.

Decoding Procedures

The application should use the following decoding procedure:

- The application can use the **MFXAudioDECODE_DecodeHeader** function to retrieve decoding initialization parameters from the bitstream. This step is optional if such parameters are retrievable from other sources such as an audio/video splitter.
- The application uses the **MFXAudioDECODE_QueryIOSize** function to obtain the recommended sizes of input and output data buffers.
- The application calls the **MFXAudioDECODE_Init** function to initialize decoder.
- The application calls the **MFXAudioDECODE_DecodeFrameAsync** function for a decoding operation. If decoding output is not available, the function returns a status code **MFX_ERR_MORE_DATA** requesting additional bitstream input.

- Upon successful decoding, the **MFXAudioDECODE_DecodeFrameAsync** function returns **MFX_ERR_NONE**. However, the decoded data is not yet available because the **MFXAudioDECODE_DecodeFrameAsync** function is asynchronous. The application must use the **MFXAudioCORE_SyncOperation** function to synchronize the decoding operation before retrieving the decoded data.

Bitstream Repositioning

The application can use the following procedure for bitstream reposition during decoding:

1. Use the **MFXAudioDECODE_Reset** function to reset the decoder.
2. Append the bitstream from the new location to the bitstream buffer.
3. Resume the decoding procedure.

Encoding Procedures

The application should use the following encoding procedure:

- The application uses the **MFXAudioENCODE_QueryIOSize** function to obtain the recommended sizes of input and output data buffers.
- The application calls the **MFXAudioENCODE_Init** function to initialize encoder.
- The application calls the **MFXAudioENCODE_EncodedFrameAsync** function for the encoding operation. Because input frame size may differ from compressed frame size, there are three possible outcomes of this call:
  - if the input buffer contains exactly one audio frame, the function starts asynchronous encoding operation and returns **MFX_ERR_NONE** status. The application should use new audio frame and new compressed bitstream buffer in next function call.
  - if the input buffer contains part of audio frame, the function does not start asynchronous encoding operation and returns **MFX_ERR_MORE_DATA** status. The application should use new audio frame and the same compressed bitstream buffer in next function call.
  - if the input buffers contains more than one audio frame, the function starts asynchronous encoding operation and returns **MFX_ERR_MORE_BITSTREAM** status. The application should use the same audio frame and new compressed bitstream buffer in next function call.
- Upon successful start of encoding operation, the function returns either **MFX_ERR_NONE** or **MFX_ERR_MORE_BITSTREAM** status. However, the encoded bitstream is not yet available because the **MFXAudioENCODE_EncodeFrameAsync** function is asynchronous. The application must use the **MFXAudioCORE_SyncOperation** function to synchronize the encoding operation before retrieving the encoded bitstream.

- At the end of the stream, the application should retrieve data cached by the encoder by continuously calling the **MFXAudioENCODE_EncodeFrameAsync** function with **NULL** pointer as input, until the function returns **MFX_ERR_MORE_DATA**.

Configuration Change

Audio does not support any dynamic configuration changes. The application should close and then reinitialize **Audio** component to change any parameters.

Transcoding Procedures

The application can use the encoding and decoding functions together for transcoding operations. This section describes the key aspects of connecting two or more functions together.

Asynchronous Pipeline

The application passes the output of an upstream **Audio** function to the input of the downstream **Audio** function to construct an asynchronous pipeline. Such pipeline construction is done at runtime and can be dynamically changed.

The **Audio** simplifies the requirement for asynchronous pipeline synchronization. The application only needs to synchronize after the last function. Explicit synchronization of intermediate results is not required and in fact can slow performance.

The **Audio** tracks the dynamic pipeline construction and verifies dependency on input and output parameters to ensure the execution order of the pipeline function. In Example 1, the **Audio** will ensure **MFXAudioENCODE_EncodeFrameAsync** does not begin its operation until **MFXAudioDECODE_DecodeFrameAsync** has finished.

During the execution of an asynchronous pipeline, the application must consider output data unavailable until the execution has finished. From the moment when the function reported successful beginning of asynchronous operation and until corresponded sync operation indicated that asynchronous operation had been completed. I.e. from the moment when **MFXAudioENCODE_EncodeFrameAsync** or **MFXAudioDECODE_DecodeFrameAsync** functions returned **ERR_NONE** status until the moment when **MFXAudioCORE_SyncOperation** completed waiting and returned **ERR_NONE** status.

The encoder can cache input audio frames and keep them in use even after correspondent output bitstream buffer has been encoded. To signal that frame is in use the encoder increases its lock counter. The application should not reuse audio frame until its lock counter will became equal to zero. It is not recommended to directly modify lock counter.

The **Audio** checks pipeline dependencies by comparing the pointers to input and output parameters of each function in the pipeline. Do not modify them before the previous asynchronous operation finishes. Doing so will break the dependency check and can result in undefined behavior.

```
mfxSyncPoint sp;
MFXAudioDECODE_DecodeFrameAsync(session, in_d, out_d, &sp_d);
MFXAudioENCODE_EncodeFrameAsync(session, out_d, out_e, &sp_e);
MFXAudioCORE_SyncOperation (session, sp_e, INFINITE);
```

Example 1: Asynchronous Pipeline

Pipeline Error Reporting

During asynchronous pipeline construction, on each stage **Audio** function will return a synchronization point (sync point). These synchronization points are useful in tracking errors during the asynchronous pipeline operation.

Assume the pipeline is **A→B→C**. The application synchronizes on sync point **C**. If the error occurs in function **C**, then the synchronization returns the exact error code. If the error occurs before function **C**, then the synchronization returns **MFX_ERR_ABORTED**. The application can then try to synchronize on sync point **B**. Similarly, if the error occurs in function **B**, the synchronization returns the exact error code, or else **MFX_ERR_ABORTED**. Same logic applies if the error occurs in function **A**.

Function Reference

This section describes **Audio** functions and their operations.

In each function description, only commonly used status codes are documented. The function may return additional status codes, such as **MFX_ERR_INVALID_HANDLE** or **MFX_ERR_NULL_PTR**, in certain case. See the **mfxStatus** enumerator for a list of all status codes.

For plugin-related functions (introduced in API 1.9) please refer to **SDK** API reference manual (`mediasdkusr-man.pdf`)

Global Functions

Global functions initialize and de-initialize the **Audio** library and perform query functions on a global scale within an application. Functions described in this chapter are common for audio and video libraries. Only audio specific functionality is described in this manual. For complete description, see "**SDK** Reference Manual".

Member Functions	Description
MFXInit	Initializes a session
MFXClose	De-initializes a session
MFXQueryIMPL	Queries the implementation type
MFXQueryVersion	Queries the implementation version
MFXJoinSession	Join two sessions together
MFXDisjoinSession	Remove the join state of the current session
MFXCloneSession	Clone the current session
MFXSetPriority	Set session priority
MFXGetPriority	Obtain session priority

MFXInit

Syntax

```
mfxStatus MFXInit(mfxIMPL impl, mfxVersion *ver, mfxSession *session);
```

Description

See "**SDK** Reference Manual".

The audio library supports only SW implementation and `impl` should be equal to **`MFX_IMPL_AUDIO | MFX_IMPL_SOFTWARE`**.

MFXClose

Syntax

```
mfxStatus MFXClose(mfxSession session);
```

Description

See "**SDK** Reference Manual".

MFXQueryIMPL

Syntax

```
mfxStatus MFXQueryIMPL(mfxSession session, mfxIMPL *impl);
```

Description

See "**SDK** Reference Manual".

MFXQueryVersion

Syntax

```
mfxStatus MFXQueryVersion(mfxSession session, mfxVersion *version);
```

Description

See "**SDK** Reference Manual".

MFXJoinSession

Syntax

```
mfxStatus MFXJoinSession(mfxSession session, mfxSession child);
```

Description

See "**SDK** Reference Manual".

The application could join several audio sessions together, but joining of audio and video sessions are not supported.

MFXDisjoinSession

Syntax

```
mfxStatus MFXDisjoinSession(mfxSession session);
```

Description

See "**SDK** Reference Manual".

MFXCloneSession

Syntax

```
mfxStatus MFXCloneSession(mfxSession session, mfxSession *clone);
```

Description

See "**SDK** Reference Manual".

MFXSetPriority

Syntax

```
mfxStatus MFXSetPriority(mfxSession session, mfxPriority priority);
```

Description

See "**SDK** Reference Manual".

MFXGetPriority

Syntax

```
mfxStatus MFXGetPriority(mfxSession session, mfxPriority *priority);
```

Description

See "**SDK** Reference Manual".

MFXAudioCORE

This class of functions consists of auxiliary functions that all functions of the implementation can call.

Member Functions	
MFXAudioCORE_SyncOperation	This function checks status or waits for completion of the given sync point and returns a status code.

MFXAudioCORE_SyncOperation

Syntax

```
mfxStatus MFXAudioCORE_SyncOperation(mfxSession session, mfxSyncPoint
syncp, mfxU32 wait);
```

Parameters

session	session handle
syncp	Sync point
wait	Wait time in milliseconds

Description

This function checks status or wait for completion of an asynchronous operation and returns the status code after the specified asynchronous operation completes. If `wait` is zero, the function returns immediately.

Return Status

MFX_ERR_NONE	The function completed successfully.
MFX_WRN_IN_EXECUTION	The specified asynchronous function is in execution.
MFX_ERR_ABORTED	The specified asynchronous function aborted due to data dependency on a previous asynchronous function that did not complete.

Remarks

See status codes for specific asynchronous functions.

MFXAudioENCODE

This class of functions performs the entire encoding process from the input audio samples to the output bitstream.

Member Functions	
MFXAudioENCODE_Query	Queries the encoder capability
MFXAudioENCODE_QueryIOSize	Queries input and output buffer sizes required for encoding
MFXAudioENCODE_Init	Initializes the encoding operation
MFXAudioENCODE_Reset	Resets the current encoding operation and prepares for the next encoding operation
MFXAudioENCODE_Close	Terminates the encoding operation and de-allocates any internal memory
MFXAudioENCODE_GetAudioParam	Obtains the current working parameter set
MFXAudioENCODE_EncodeFrameAsync	Performs the encoding and returns the compressed bitstream

MFXAudioENCODE_Query

Syntax

```
mfxStatus MFXAudioENCODE_Query(mfxSession session, mfxAudioParam *in,
mfxAudioParam *out);
```

Parameters

session	session handle
in	Pointer to the **mfxAudioParam** structure as input
out	Pointer to the **mfxAudioParam** structure as output

Description

This function works in either of two modes:

1. If the `in` pointer is zero, the function returns the class configurability in the output **mfxAudioParam** structure. A non-zero value in each field of the output structure indicates that the implementation can configure the field with **Init**.
2. If the `in` parameter is non-zero, the function checks the validity of the fields in the input **mfxAudioParam** structure. Then the function returns the corrected values in the output **mfxAudioParam** structure. If there is insufficient information to determine the validity or correction is impossible, the function zeroes the fields. This feature can verify whether the implementation supports certain profiles, levels or bitrates.

The application can call this function before or after it initializes the encoder. The **CodecId** field of the output **mfxAudioParam** structure is a mandated field (to be filled by the application) to identify the coding standard.

Return Status

MFX_ERR_NONE	The function completed successfully.
MFX_ERR_UNSUPPORTED	The function failed to identify a specific implementation for the required features.
MFX_WRN_INCOMPATIBLE_AUDIO_PARAM	The function detected some audio parameters were incompatible with others; incompatibility resolved.

MFXAudioENCODE_QueryIOSize

Syntax

```
mfxStatus MFXAudioENCODE_QueryIOSize(mfxSession session, mfxAudioParam
*par, mfxAudioAllocRequest *request);
```

Parameters

session	session handle
par	Pointer to the **mfxAudioParam** structure as input
request	Pointer to the **mfxAudioAllocRequest** structure as output

Description

This function returns input and output buffer sizes required for encoding.

The CodecId field of the **mfxAudioParam** structure is a mandated field (to be filled by the application) to identify the coding standard.

This function does not validate I/O parameters except those used in calculating of the buffer sizes.

Return Status

MFX_ERR_NONE	The function completed successfully.
MFX_ERR_INVALID_AUDIO_PARAM	The function detected invalid audio parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.
MFX_WRN_INCOMPATIBLE_AUDIO_PARAM	The function detected some audio parameters were incompatible with others; incompatibility resolved.

MFXAudioENCODE_Init

Syntax

```
mfxStatus MFXAudioENCODE_Init(mfxSession session, mfxAudioParam *par);
```

Parameters

session	session handle

par	Pointer to the **mfxAudioParam** structure

Description

This function allocates memory and initializes the encoder. This function also does extensive validation to ensure if the configuration, as specified in the input parameters, is supported.

Return Status

MFX_ERR_NONE	The function completed successfully.
MFX_ERR_INVALID_AUDIO_PARAM	The function detected invalid audio parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.
MFX_WRN_INCOMPATIBLE_AUDIO_PARAM	The function detected some audio parameters were incompatible with others; incompatibility resolved.
MFX_ERR_UNDEFINED_BEHAVIOR	The function is called twice without a close

MFXAudioENCODE_Reset

Syntax

```
mfxStatus MFXAudioENCODE_Reset(mfxSession session, mfxAudioParam *par);
```

Parameters

session	session handle
par	Pointer to the **mfxAudioParam** structure

Description

This function stops the current encoding operation and restores internal structures or parameters for a new encoding operation, possibly with new parameters.

Return Status

MFX_ERR_NONE	The function completed successfully.

MFX_ERR_INVALID_AUDIO_PARAM	The function detected that audio parameters are wrong or they conflict with initialization parameters. Reset is impossible.
MFX_ERR_INCOMPATIBLE_AUDIO_PARAM	The function detected that provided by the application audio parameters are incompatible with initialization parameters. Reset requires additional memory allocation and cannot be executed. The application should close the component and then reinitialize it.
MFX_WRN_INCOMPATIBLE_AUDIO_PARAM	The function detected some audio parameters were incompatible with others; incompatibility resolved.

MFXAudioENCODE_Close

Syntax

```
mfxStatus MFXAudioENCODE_Close(mfxSession session);
```

Parameters

session	session handle

Description

This function terminates the current encoding operation and de-allocates any internal tables or structures.

Return Status

MFX_ERR_NONE	The function completed successfully.

MFXAudioENCODE_GetAudioParam

Syntax

```
mfxStatus MFXAudioENCODE_GetAudioParam(mfxSession session, mfxAudioParam
*par);
```

Parameters

session	session handle
par	Pointer to the corresponding parameter structure

Description

This function retrieves current working parameters to the specified output structure. If extended buffers are to be returned, the application must allocate those extended buffers and attach them as part of the output structure.

Returned information

MFX_ERR_NONE	The function completed successfully.

MFXAudioENCODE_EncodeFrameAsync

Syntax

```
mfxStatus MFXAudioENCODE_EncodeFrameAsync(mfxSession session,
mfxAudioFrame *frame, mfxBitstream *bs, mfxSyncPoint *syncp);
```

Parameters

Session	Session handle
frame	Pointer to input audio frame.
bs	Pointer to the output compressed bitstream.
syncp	Pointer to the returned sync point associated with this operation.

Description

This function takes input audio samples and encodes them in compressed bitstream.

The application should provide new output buffer for each compressed frame. I.e. each sync operation should correspond to separate output buffer. It is not required to provide empty data buffer as output, but the application should ensure that there is sufficient space in the output buffer. The function **MFXAudioENCODE_QueryIOSize**

returns required output buffer sizes.

This function is asynchronous.

See Encoding Procedures and Asynchronous Pipeline for more details.

Return Status

MFX_ERR_NONE	The function completed successfully.
MFX_ERR_MORE_DATA	The function requires more data to generate any output.
MFX_ERR_MORE_BITSTREAM	The function requires more bitstream buffers to store output.
MFX_ERR_NOT_ENOUGH_BUFFER	The output bitstream buffer size is insufficient.

MFXAudioDECODE

This class of functions implements a complete decoder that decompresses input bitstream to audio samples.

Member Functions	
MFXAudioDECODE_Query	Queries the feature capability
MFXAudioDECODE_DecodeHeader	Parses the bitstream to obtain the audio parameters for initialization
MFXAudioDECODE_Init	Initializes the decoding operation
MFXAudioDECODE_Reset	Resets the current decoding operation and prepares for the next decoding operation
MFXAudioDECODE_Close	Terminates the decoding operation and de-allocates any internal memory
MFXAudioDECODE_QueryIOSize	Queries the number of frames required for decoding

MFXAudioDECODE_GetAudioParam	Obtains the current working parameter set
MFXAudioDECODE_DecodeFrameAsync	Performs decoding from the input bitstream to the output frame surface

MFXAudioDECODE_Query

Syntax

```
mfxStatus MFXAudioDECODE_Query(mfxSession session, mfxAudioParam *in,
mfxAudioParam *out);
```

Parameters

session	Session handle
in	Pointer to the **mfxAudioParam** structure as input
out	Pointer to the **mfxAudioParam** structure as output

Description

This function works in one of two modes:

1. If the `in` pointer is zero, the function returns the class configurability in the output **mfxAudioParam** structure. A non-zero value in each field of the output structure indicates that the field is configurable by the implementation with the **MFXAudioDECODE_Init** function).
2. If the `in` parameter is non-zero, the function checks the validity of the fields in the input **mfxAudioParam** structure. Then the function returns the corrected values to the output **mfxAudioParam** structure. If there is insufficient information to determine the validity or correction is impossible, the function zeros the fields. This feature can verify whether the implementation supports certain profiles, levels or bitrates.

The application can call this function before or after it initializes the decoder. The `CodecId` field of the output **mfxAudioParam** structure is a mandated field (to be filled by the application) to identify the coding standard.

Return Status

MFX_ERR_NONE	The function completed successfully.
MFX_ERR_UNSUPPORTED	The function failed to identify a specific implementation.
MFX_WRN_INCOMPATIBLE_AUDIO_PARAM	The function detected some audio parameters were incompatible with others; incompatibility resolved.

MFXAudioDECODE_QueryIOSize

Syntax

```
mfxStatus MFXAudioDECODE_QueryIOSurf(mfxSession session, mfxAudioParam
*par, mfxAudioAllocRequest *request);
```

Parameters

session	session handle
par	Pointer to the **mfxAudioParam** structure as input
request	Pointer to the **mfxAudioAllocRequest** structure as output

Description

This function returns input and output buffer sizes required for decoding.

The CodecId field of the **mfxAudioParam** structure is a mandated field (to be filled by the application) to identify the coding standard.

This function does not validate I/O parameters except those used in calculating of the buffer sizes.

Return Status

MFX_ERR_NONE	The function completed successfully.
MFX_ERR_INVALID_AUDIO_PARAM	The function detected invalid audio parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.
MFX_WRN_INCOMPATIBLE_ AUDIO _PARAM	The function detected some audio parameters were incompatible with others; incompatibility resolved.

MFXAudioDECODE_DecodeHeader

Syntax

```
mfxStatus MFXAudioDECODE_DecodeHeader(mfxSession session, mfxBitstream
*bs, mfxAudioParam *par);
```

Parameters

session	session handle
bs	Pointer to the bitstream
par	Pointer to the **mfxAudioParam** structure

Description

This function parses the input bitstream and fills the **mfxAudioParam** structure with appropriate values, such as number of channels and sample frequency, for the **Init** function. The application can then pass the resulting **mfxAudioParam** structure to the **MFXAudioDECODE_Init** function for decoder initialization.

An application can call this function at any time before or after decoder initialization.

The CodecId field of the **mfxAudioParam** structure is a mandated field (to be filled by the application) to identify the coding standard.

Return Status

MFX_ERR_NONE	The function successfully filled **mfxAudioParam** structure. It does not mean that the stream can be decoded by the **Audio**. The application should call **MFXAudioDECODE_Query** function to check if decoding of the stream is supported.
MFX_ERR_MORE_DATA	The function requires more bitstream data.

MFXAudioDECODE_Init

Syntax

```
mfxStatus MFXAudioDECODE_Init(mfxSession session, mfxAudioParam *par);
```

Parameters

session	session handle
par	Pointer to the **mfxAudioParam** structure

Description

This function allocates memory and initializes the decoder. This function also does extensive validation to determine whether the configuration is supported as specified in the input parameters.

Return Status

MFX_ERR_NONE	The function completed successfully.
MFX_ERR_INVALID_AUDIO_PARAM	The function detected invalid audio parameters. These parameters may be out of the valid range, or the combination of parameters resulted in an incompatibility error. Incompatibility was not resolved.
MFX_WRN_INCOMPATIBLE_AUDIO_PARAM	The function detected some audio parameters were incompatible; Incompatibility resolved.
MFX_ERR_UNDEFINED_BEHAVIOR	The function is called twice without a close.

MFXAudioDECODE_Reset

Syntax

```
mfxStatus MFXAudioDECODE_Reset(mfxSession session, mfxAudioParam *par);
```

Parameters

session	session handle
par	Pointer to the **mfxAudioParam** structure

Description

This function stops the current decoding operation and restores internal structures or parameters for a new decoding operation.

Reset serves two purposes:

- It recovers the decoder from errors.

- It restarts decoding from a new position.

Return Status

MFX_ERR_NONE	The function completed successfully.
MFX_ERR_INVALID_AUDIO_PARAM	The function detected that audio parameters are wrong or they conflict with initialization parameters. Reset is impossible.
MFX_ERR_INCOMPATIBLE_ AUDIO _PARAM	The function detected that provided by the application audio parameters are incompatible with initialization parameters. Reset requires additional memory allocation and cannot be executed. The application should close the component and then reinitialize it.
MFX_WRN_INCOMPATIBLE_ AUDIO _PARAM	The function detected some audio parameters were incompatible; Incompatibility resolved.

MFXAudioDECODE_Close

Syntax

```
mfxStatus MFXAudioDECODE_Close(mfxSession session);
```

Parameters

session	session handle

Description

This function terminates the current decoding operation and de-allocates any internal tables or structures.

Return Status

MFX_ERR_NONE	The function completed successfully.

MFXAudioDECODE_GetAudioParam

Syntax

```
mfxStatus MFXAudioDECODE_GetAudioParam(mfxSession session, mfxAudioParam
*par);
```

Parameters

session	session handle
par	Pointer to the corresponding parameter structure

Description

This function retrieves current working parameters to the specified output structure. If extended buffers are to be returned, the application must allocate those extended buffers and attach them as part of the output structure.

Return Status

MFX_ERR_NONE	The function completed successfully.

MFXAudioDECODE_DecodeFrameAsync

Syntax

```
mfxStatus MFXAudioDECODE_DecodeFrameAsync(mfxSession session,
mfxBitstream *bs, mfxAudioFrame *frame, mfxSyncPoint *syncp);
```

Parameters

session	session handle
bs	Pointer to the compressed bitstream
frame	Pointer to the buffer containing decoded audio frame
syncp	Pointer to the sync point associated with this operation

Description

This function decodes the compressed bitstream to the raw audio samples.

Depending on audio standard, the decoder accepts different amount of data as input. For AAC it should be exactly one frame. For MP3 it may be part of frame, complete frame or several frames. If there is not enough data to decode an audio frame, the function returns **MFX_ERR_MORE_DATA**, and consumes all input bits except the case when a start code or header is located at the end of the buffer. In this case, the function leaves the last few bytes in the bitstream buffer. If there is more incoming bitstream, the application should append the incoming bitstream to the bitstream buffer.

If the application appends additional data to the bitstream buffer, it is possible that the bitstream buffer will contain more than one frame. It is recommended that the application invoke the function repeatedly until the function returns **MFX_ERR_MORE_DATA**, before appending any more data to the bitstream buffer.

The application should provide separate output buffer for each audio frame. I.e. each sync operation should correspond to separate output buffer. It is not required to provide empty data buffer as output, but the application should ensure that there is sufficient space in the output buffer. The function **MFXAudioDECODE_QueryIOSize** returns required output buffer sizes.

If function has successfully started asynchronous decoding, it returns **MFX_ERR_NONE** status and fills in output audio frame. The application can immediately access output audio frame to read time stamp, data size, number of channels and other information. However, the application should not access actual audio samples until decoding is finished.

This function is asynchronous.

See Decoding Procedures and Asynchronous Pipeline for more details.

Return Status

MFX_ERR_NONE	The function completed successfully and the output bitstream is ready for decoding.
MFX_ERR_MORE_DATA	The function requires more bitstream at input before decoding can proceed.

Structure Reference

In the following structure references, all reserved fields must be zero.

mfxVersion

Definition

```
typedef union _mfxVersion {
      struct {
            mfxU16      Minor;
            mfxU16      Major;
            };
      mfxU32      Version;
} mfxVersion;
```

Description

See "**SDK** Reference Manual".

mfxBitstream

Definition

```
typedef struct {
     union {
        struct {
            mfxEncryptedData* EncryptedData;
            mfxExtBuffer **ExtParam;
            mfxU16  NumExtParam;
        };
         mfxU32  reserved[6];
     };
    mfxI64  DecodeTimeStamp;
    mfxU64  TimeStamp;
    mfxU8*  Data;
```

```
    mfxU32  DataOffset;
    mfxU32  DataLength;
    mfxU32  MaxLength;

    mfxU16  PicStruct;
    mfxU16  FrameType;
    mfxU16  DataFlag;
    mfxU16  reserved2;
} mfxBitstream;
```

Description

The `mfxBitstream` structure defines the buffer that holds compressed audio bitstream.

Reserved fields either intended for future extension or have no meaning for audio data.

Members

`EncryptedData`	Reserved and must be zero.
`ExtParam`	Reserved and must be zero.
`NumExtParam`	Reserved and must be zero.
`DecodeTimeStamp`	Reserved and must be zero.
`TimeStamp`	Time stamp of the compressed bitstream or audio samples in units of 90KHz. A value of **`MFX_TIMESTAMP_UNKNOWN`** indicates that there is no time stamp.
`Data`	Bitstream buffer pointer—32-bytes aligned
`DataOffset`	Next reading or writing position in the bitstream buffer
`DataLength`	Size of the actual bitstream data in bytes
`MaxLength`	Allocated bitstream buffer size in bytes
`PicStruct`	Reserved and must be zero.
`FrameType`	Reserved and must be zero.
`DataFlag`	Reserved and must be zero.

mfxAudioAllocRequest

Definition

```
typedef struct {
    mfxU32  SuggestedInputSize;
    mfxU32  SuggestedOutputSize;
    mfxU32  reserved[6];
} mfxAudioAllocRequest;
```

Description

The `mfxAudioAllocRequest` structure describes buffer sizes required for decoding and encoding. These are minimum required numbers. The application may allocate bigger buffers.

Members

`SuggestedInputSize`	Suggested input buffer size in byte.
`SuggestedOutputSize`	Suggested output buffer size in byte.

mfxAudioInfoMFX

Definition

```
typedef struct {
    mfxU32  CodecId;
    mfxU16  CodecProfile;
    mfxU16  CodecLevel;

    mfxU32  Bitrate;
    mfxU32  SampleFrequency;
    mfxU16  NumChannel;
```

```
    mfxU16  BitPerSample;

    mfxU16  reserved1[22];

    union {
        struct {    /* AAC Decoding Options */
            mfxU16      Layer;
            mfxU16      reserved2[14];
            mfxU16      AACHeaderDataSize;
            mfxU8       AACHeaderData[64];
        };
        struct {    /* AAC Encoding Options */
            mfxU16      OutputFormat;
            mfxU16      StereoMode;
        };
    };
} mfxAudioInfoMFX;
```

Description

This structure specifies configurations for decoding and encoding processes.

Members

`CodecId`	Specifies the codec format identifier in the FOURCC code; see the **`CodecFormatFourCC`** enumerator for details. This is a mandated input parameter for **`Query`**, **`QueryIOSize`** and **`Init`** functions.
`CodecProfile`	Specifies the codec profile; see the **`CodecProfile`** enumerator for details. Specify the codec profile explicitly or the functions will determine the correct profile from other sources.
`CodecLevel`	Codec level; see the **`CodecLevel`** enumerator for details. Specify the codec level explicitly or the functions will determine the correct level from other sources.
`Bitrate`	Bitrate of compressed audio stream in bits per second. It may be arbitrary value for AAC and one of the predefined by standard values for MP3.
`SampleFrequency`	Sample frequency of audio data.
`NumChannel`	Number of channels in bitstream.
`BitPerSample`	Number of bits per audio sample.

	AAC decoding options
`Layer`	Audio layer. It is not set by **`MFXAudioDECODE_DecodeHeader`** function and should be set by application.
`AACHeaderDataSize`	ADIF or ADTS or ESDS header size.
`AACHeaderData[64]`	ADIF or ADTS or ESDS header. It is mandatory as input parameter for bitstreams extracted from MP4 container.
	AAC encoding options
`OutputFormat`	Specifies header type. It is one of the next values: • `MFX_AUDIO_AAC_ADTS` – use ADTS header • `MFX_AUDIO_AAC_ADIF` – use ADIF header • `MFX_AUDIO_AAC_RAW` – don't add header to compressed bitstream
`StereoMode`	Specifies stereo mode. It is one of the next values: • `MFX_AUDIO_AAC_MONO` – encode as mono • `MFX_AUDIO_AAC_LR_STEREO` – encode as two separate channels • `MFX_AUDIO_AAC_MS_STEREO` – encodes as sum and difference of stereo channels • `MFX_AUDIO_AAC_JOINT_STEREO` - encode as joint stereo

mfxAudioParam

Definition

```
typedef struct {
    mfxU16  AsyncDepth;
    mfxU16  Protected;
    mfxU16  reserved[14];

    mfxAudioInfoMFX   mfx;
    mfxExtBuffer**    ExtParam;
```

```
    mfxU16              NumExtParam;
} mfxAudioParam;
```

Description

The `mfxAudioParam` structure contains configuration parameters for encoding, decoding and transcoding.

Members

`AsyncDepth`	Specifies how many asynchronous operations an application performs before the application explicitly synchronizes the result. If zero, the value is not specified.
`Protected`	Reserved and must be zero.
`mfx`	Configurations related to encoding, decoding and transcoding; see the definition of the **`mfxAudioInfoMFX`** structure for details.
`NumExtParam`	Reserved and must be zero.
`ExtParam`	Reserved and must be zero.

mfxAudioFrame

Definition

```
typedef struct {
    mfxU64  TimeStamp;
    mfxU16  Locked;
    mfxU16  NumChannel;
    mfxU32  SampleFrequency;
    mfxU16  BitPerSample;
    mfxU16  reserved1[7];

    mfxU8*  Data;
    mfxU32  reserved2;
```

```
    mfxU32  DataLength;
    mfxU32  MaxLength;

    mfxU32  NumExtParam;
    mfxExtBuffer **ExtParam;
} mfxAudioFrame;
```

Description

The `mfxAudioFrame` structure defines buffer that holds raw audio samples. It is used to store decoder output or encoder input.

Members

`TimeStamp`	Time stamp of the audio frame in units of 90KHz. A value of **`MFX_TIMESTAMP_UNKNOWN`** indicates that there is no time stamp.
`Locked`	Lock counter. If this field is greater than zero, then audio frame is used by the **Audio** and the application should not access its content. It is not recommended to directly change this field.
`NumChannel`	Number of audio channels in buffer
`SampleFrequency`	Sample frequency of audio data in buffer
`BitPerSample`	Number of bits per audio sample
`Data`	Pointer to data buffer
`DataLength`	Size of the actual audio data in bytes
`MaxLength`	Allocated data buffer size in bytes
`NumExtParam`	Reserved and must be zero.
`ExtParam`	Reserved and must be zero.

Enumerator Reference

CodecFormatFourCC

Description

The `CodecFormatFourCC` enumerator itemizes codecs in the FourCC format.

Name/Description

`MFX_CODEC_AAC`

`MFX_CODEC_MP3`

CodecProfile

Description

The `CodecProfile` enumerator itemizes codec profiles for all codecs.

Name/Description

`MFX_PROFILE_UNKNOWN`	Unspecified profile

`MFX_PROFILE_AAC_LC`
`MFX_PROFILE_AAC_LTP`
`MFX_PROFILE_AAC_MAIN`
`MFX_PROFILE_AAC_SSR`
`MFX_PROFILE_AAC_HE`
`MFX_PROFILE_AAC_ALS`
`MFX_PROFILE_AAC_BSAC`
`MFX_PROFILE_AAC_PS`
	AAC profiles:
	• LC - low complexity
	• LTP - long term prediction
	• MAIN - main
	• SSR - scalable sample rate
	• HE - high efficiency
	• ALS - audio lossless coding
	• BSAC - bit slice arithmetic coding
	• PS - parametric stereo
	
	Different implementation of the **Audio** library may support different sets of profiles. The application has to use Query function to determine if particular profile is supported.

	Generally, AAC decoder supports all profiles specified above. Encoder usually supports LC, LTP, HE, PS and MAIN profiles.
MFX_MPEG1_LAYER1_AUDIO MFX_MPEG1_LAYER2_AUDIO MFX_MPEG1_LAYER3_AUDIO MFX_MPEG2_LAYER1_AUDIO MFX_MPEG2_LAYER2_AUDIO MFX_MPEG2_LAYER3_AUDIO	MP3 layers.

CodingOptionValue

Description

See "**SDK** Reference Manual".

Corruption

Description

See "**SDK** Reference Manual". The audio decoders support next values:

Name/Description

MFX_CORRUPTION_MINOR	Minor corruption in decoding certain audio samples
MFX_CORRUPTION_MAJOR	Major corruption in decoding the frame

mfxIMPL

Description

See "**SDK** Reference Manual". The audio library supports only next values:

Name/Description

MFX_IMPL_SOFTWARE	Use the software implementation
MFX_IMPL_AUDIO	Load audio library. It can be used only together with MFX_IMPL_SOFTWARE, any other combinations lead to error.

mfxPriority

Description

See "**SDK** Reference Manual".

mfxStatus

Description

See "**SDK** Reference Manual" for complete list of statuses. The audio library may returns most of those statuses plus several audio specific described below.

Name/Description

MFX_ERR_INVALID_AUDIO_PARAM	Invalid audio parameters detected. **Init** and **Reset** functions return this status code to indicate either that mandated input parameters are unspecified, or the functions failed to correct them.
MFX_ERR_INCOMPATIBLE_AUDIO_PARAM	Incompatible audio parameters detected. If a **Reset** function returns this status code, a component—decoder or encoder — cannot process the specified configuration with existing structures and buffers. If the function **MFXAudioDECODE_DecodeFrameAsync** returns this status code, the bitstream contains an incompatible audio parameter configuration that the decoder cannot follow.
MFX_WRN_INCOMPATIBLE_A	Incompatible audio parameters detected. Functions return

UDIO_PARAM	this status code to indicate that there was incompatibility in the specified parameters and has resolved it.